Amphion/NAV: Deductive Synthesis of State Estimation Software

Jon Whittle, QSS Inc.; Jeffrey Van Baalen, U. Wyoming; Johann Schumann, RIACS

Peter Robinson, QSS Inc.; Tom Pressburger, NASA; John Penix, NASA

Phil Oh, QSS Inc.; Mike Lowry, NASA; Guillaume Brat, Kestrel Inst.

NASA Ames Research Center
Moffett Field, CA 94035

Abstract

This paper describes technology developed for the Amphion/NAV synthesis system. Building on previous work in Amphion/NAIF, this system synthesizes graduate-level textbook examples of single-mode geometric state estimation software. Amphion/NAV includes explanation technology for mapping the internal representations of a proof (generated through deductive synthesis) that a program is correct, to documentation and requirements traces that demonstrate this correctness to human engineers. The ExplainIt! subsystem extends the technology previously reported on for explanation generation, and includes an XML-based browser interface. A methodology that employs multiple levels of reuse was used in developing Amphion/NAV, including incorporation of the previous Amphion/NAIF domain theory as a part of the Amphion/NAV domain theory. The combination of general-purpose automated reasoning methods that are widely applicable, with specialized automated reasoning methods that are domain specific, enabled generation of substantial programs. Lessons learned are presented, and future work is discussed.

1 Introduction

Previous work on domain-specific deductive program synthesis [12, 13] described the Amphion/NAIF system for generating Fortran code from high-level graphical specifications describing problems in space system geometry. Amphion/NAIF specifications describe functions that compute geometric quantaties (e.g., the distance between two planets at some point in time) by composing together Fortran subroutines from the NAIF subroutine library developed at the Jet Propulsion Laboratory (JPL). In essence, Amphion/NAIF synthesizes code for glueing together the NAIF components in a way such that the generated code implements the specification. The correctness of this implementation (with respect to the specification) is guaranteed by the use of the SNARK first-order resolution theorem prover to carry out the refinement from specification to a functional λ-term, and the use of a transformation system to transform the λ-term into Fortran code.

Amphion/NAIF demonstrated the success of domain-specific deductive program synthesis and is still in use today within the space science community. However, a number of questions remained open that we will attempt to answer in this paper, namely:

- What evidence can be provided to an interested party that the generated code does in fact implement the specification? Note that the use of a theorem prover in synthesis is not the final answer since a prover such as SNARK is a complex software artifact in itself and may contain bugs. Amphion/NAIF did not address the integration of a program synthesis system into the software process - particularly the issue of validation and verification dependent on human review.
- What are the issues involved for a project team to develop a program synthesis system for its own particular domain? Amphion/NAIF generates relatively simple code, principally sequences of subroutine calls with limited use of loops. In developing program synthesis systems for other domains (e.g., for embedded systems applications), these restrictions will no

longer hold. In addition, Amphion/NAIF was able to build on a well-defined, previously given component library. Such a library is unlikely to be available for most applications, but must be developed in parallel or collected from elsewhere.

In order to investigate these two limitations, we developed Amphion for a new, much richer domain, namely Guidance, Navigation & Control (GN&C) algorithms, in particular single-mode geometric state estimation software. GN&C algorithms are often complex, involving iterative loop algorithms and real-time considerations such as extrapolating sensor data so that data is integrated at the same point in time. In addition, although there are standard components available for this domain (e.g., matrix manipulations, Kalman filter algorithms), there is no easily defined set of components that cover the domain fully.

Amphion/NAV is an extension of Amphion/NAIF that generates code for integrating data from multiple sensor sources in a statistically optimal way using Kalman filters [1, 4]. In addition to dealing with a much richer domain, Amphion/NAV significantly extends the explanation capabilities of the previous Amphion system [13]. The code generated by Amphion/NAV is annotated with detailed explanations describing where each expression in the code came from. These explanations are constructed by tracing automatically through the proof that produced the code and composing explanations for each of the axioms used in the proof. As a result, each program expression can be explained in terms of the concepts in the specification from which they were derived. These explanations are given in the form of hyper-linked text and are specific to the GN&C domain.

Detailed explanations provide a means for a certification body such as the FAA to examine the code in detail and to know precisely where each code expression came from. GN&C algorithms are often used in safety-critical systems. Hence the very detailed explanations provided by Amphion/NAV provide necessary documentation for the certification process. The explanations are also crucial to programmers who need to modify the generated code or integrate it into a larger system.

The domain of state estimation turns out to be a good challenge domain for deductive synthesis. Developing state estimation software tends to be a black art. In principle, the engineer should develop a mathematical model that closely resembles the real-world characteristics of the problem. The output of simulation runs on this model should then be used to refine the model until a threshold level of accuracy is reached. In practice, however, engineers start off with a mathematical model but the time and cost constraints associated with the project mean that they merely "tweak" parameters in their code rather than reassessing the fidelity of the model. Program synthesis encourages analysis to take place at the modeling level and enables rapid design space exploration.

In summary, AMPHION/NAV is a working prototype that makes the following contributions to the field of program synthesis:

- generation of complex software artifacts involving iterative loops and real-time aspects;
- generation of detailed explanations that provide documentation necessary for the certification process of safety critical systems;
- the use of a theorem prover to guarantee domain-specific properties in the generated code (as a by-product of synthesis) — e.g., all parameters are in the same frame/coordinate system, matrix multiplications are well-defined;
- rapid design space exploration of competing architectures/configurations.

2 Background on State Estimation

The domain of interest for Amphion/NAV is that of geometric state estimation, i.e., estimating the actual values of certain *state variables* (such as position, velocity, attitude) based on noisy data from multiple sensor sources. The standard technique for integrating multiple sensor data is to use a Kalman filter. A Kalman filter estimates the state of a linear dynamic system perturbed by Gaussian white noise using measurements linearly related to the state but also corrupted by Gaussian white noise. A Kalman filter provides a statistically optimal estimate, in the sense that it minimizes the expected risk associated with any quadratic loss function of the estimation error. The Kalman filter algorithm is essentially a recursive version of linear least squares with incremental updates.

The state estimation problem can be represented by the following equations, given in vector form:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), t) + \mathbf{w}(t) \tag{1}$$

$$\mathbf{z}(t) = h(\mathbf{x}(t), t) + \mathbf{v}(t) \tag{2}$$

The first equation is the process model, a vector differential equation modeling how the state vector, $\mathbf{x}(t)$, changes over time. The second equation is the measurement model, relating the measured variables to the state variables. Specifically, $\mathbf{x}(t)$ is the state vector (with $\dot{\mathbf{x}}(t)$ the time derivative) of quantities to be estimated (e.g. position, attitude etc.), $\mathbf{z}(t)$ is a vector of measurements (the state variables are not necessarily measured directly), $\mathbf{v}(t)$, $\mathbf{w}(t)$ are Gaussian white noise perturbances on the measurement and process model respectively and f and h are possibly nonlinear continuous functions that must be discretized for implementation purposes.

A Kalman filter is an iterative algorithm that returns a time sequence of estimates of the state vector, $\hat{\mathbf{x}}(t)$, by fusing the measurements with estimates of the state variables based on the process model in an optimal fashion. The estimates minimize the mean-square estimation error. In the case where either f or h is nonlinear, a Kalman filter can still be used by first linearizing around a "nominal" estimate. After linearization and discretization, f and h can be represented by matrices Φ (the state transition matrix) and H (the measurement matrix) respectively.

The standard implementation of a Kalman filter requires seven inputs: the Φ and H matrices, the covariance structure of the process and measurement noise ($\mathbf{w}(t)$ and $\mathbf{v}(t)$), an initial state estimate $\mathbf{x}(t_0)$, an error covariance matrix of the initial estimate and, of course, the sequence of measurements. During each iteration of the filter, the state estimate is updated based on new measurements and the estimate error covariance is updated for the next iteration.

To illustrate, consider an aircraft that is equipped with an inertial navigation system (INS) and radio equipment to determine the distance of the aircraft from two fixed radio towers (distance measuring equipment (DME) sensors). An inertial navigation system (for detail see e.g., [10]) is an electromechanical device which consists of three gyros and accelerometers oriented with respect to the x, y, and z axis of a rectangular coordinate system. Each movement of the INS (and the aircraft) generates a deviation signal in the gyros which, together with readings from the accelerometers, provide an estimate of the current position, velocity and attitude of the aircraft. However, due to inherent errors of the INS (principally, gyro drift), the accuracy of the estimate decreases considerably over time. Therefore, additional (or *aiding*) measurements are needed to contribute information and thus bound the error. In our example, two DME sensors provide aiding measurements in the x, y plane.

Figure 1 gives the AMPHION/NAV graphical specification for integrating measurements from an INS sensor with those from two DME sensors using a Kalman filter[1]. Figure 2A shows results from a simulation using code synthesized automatically from this specification. The simulation models the aircraft as flying in the north-south direction, but under turbulence so it is not flying in a straight line, but in a "random-walk" like fashion. The location of the two radio towers are marked by "+". The dashed line shows the estimated (x, y) position of the aircraft, based upon the INS system aided by the two DME readings.

The relative error of the Kalman filter estimate is shown

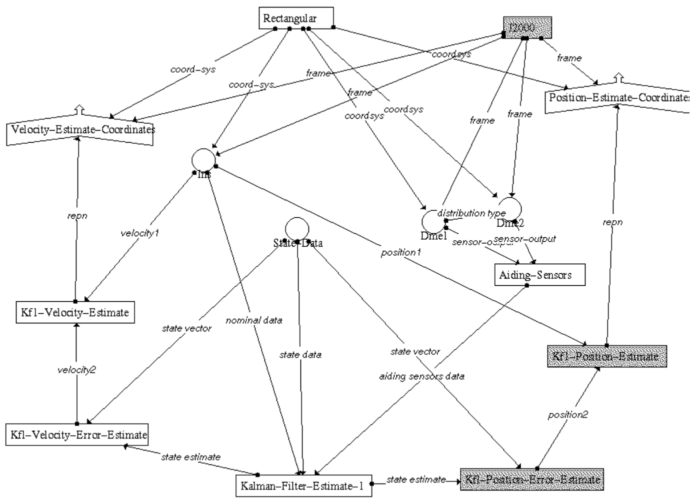


Figure 1. Graphical input specification for a simple INS configuration with two aiding DME sensors.

in Figure 2B. The error in the vertical direction (the vertical scale is logarithmic) grows in an unbounded fashion (dot-dot-dash), because the aiding sensors are only contributing to the horizontal x and y directions. After a short time, all estimates in the (vertical) z direction become practically useless.

A typical way to avoid this problem is to add a sensor which provides information about the altitude of the aircraft (e.g., a baro-altimeter). With traditional software development of the state-estimation software, adding a sensor in this way leads to a lengthy re-implementation of much of the code. Within AMPHION/NAV, a sensor can be added easily (within about 5 minutes). AMPHION/NAV then synthesizes code for the new sensor configuration. Figure 2C shows the relative errors for the new configuration. The errors in the x and y direction remain the same (the altimeter does not contribute to that); however, the vertical error is reduced substantially.

3 AMPHION/NAV System Architecture

Figure 3 presents the architecture of the AMPHION/NAV system. The domain theory (Section 4) specifies the types and operation signatures in the domain, and axioms describing the implementation of the *abstract* operations (which are used in the problem specification) in terms of *concrete* operations (which are used in the implementation). The domain theory also contains explanation templates associ-

[1] We assume that all sensor readings, measurements, and position estimates are in the same coordinate system and frame and the same units. AMPHION/NAV, however, can automatically generate code for the appropriate conversion routines.

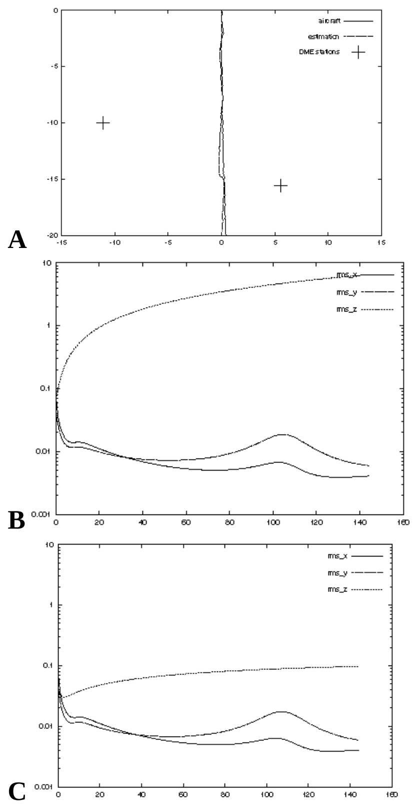


Figure 2. Simulation results for synthesized code: actual and estimated aircraft position on the x, y plane (A), relative errors in the configuration INS with two DME sensors (B), and with an additional baro-altimeter (C).

ated with each axiom. The templates provide documentation about the meaning of the axioms.

A GUI (not shown), guided by the types and operation signatures specified in the domain theory, aids the user in producing a formal, type-correct graphical problem specification (Figure 1 for an example). The specification is equivalent to a first-order logic formula of the form

$$\forall(\textit{inputs})\ \exists(\textit{outputs})\ \exists(\textit{intermediates})$$
$$(\textit{Conjunct}_1 \wedge \ldots \wedge \textit{Conjunct}_n).$$

The conjuncts describe the desired input/output relationship. The conjuncts are all expressed in terms of the abstract language, except for those expressing the relationships between concrete input or output variables and the abstract variables they represent.

The process of *deductive synthesis* [6, 9] submits the specification and the axioms of the domain theory to the synthesis engine, which is the SNARK [11] refutation-based theorem prover. The theorem prover proves that the specification is a consequence of the domain theory, and returns a proof and a vector of witness terms for the output variables, in our case an applicative term comprising the synthesized program.

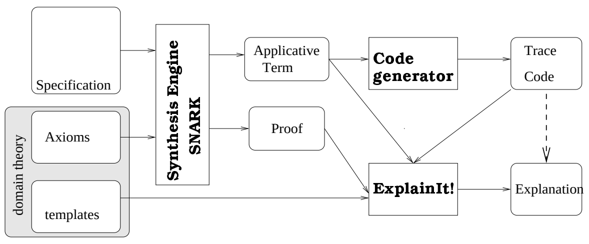


Figure 3. Architecture of AMPHION/NAV with ExplainIt!

The code generator is given the applicative term and produces code in the target programming language via application of several phases of program transformations. The target language in AMPHION/NAV is C++ and OCTAVE [2]. AMPHION/NAIF generated FORTRAN code, but only the last phase of the code generator needed to be changed for AMPHION/NAV.

The code generator records a trace of the application of the transformations. The ExplainIt! component (Section 5) accepts the axiom explanation templates, the proof, the applicative term, and the code generator trace, and produces an explanation structure for the final code. This structure links portions of the target code to explanations. The explanation of a portion of target code is generated from the explanation templates associated with the axioms that were used in the creation of that portion of the code.

4 Engineering a Domain Theory for State Estimation

Designing domain theories for program synthesis is a difficult task. Domain knowledge is often ill-defined or distributed among multiple sources of knowledge, e.g. domain experts. In addition, the scope of the domain for a particular application is usually very fuzzy (cf., for example, [2]). This problem is further complicated for supporting program synthesis because domain knowledge represents both the operations and algorithms in the domain and how those domain elements are properly applied. As a result, domain engineering for program synthesis is a significantly more difficult task than programming.

For the initial version of AMPHION/NAV described in

[2]A Matlab clone: `http://www.octave.org`

this paper, the scope is that of graduate-level textbook (e.g., [1]) state estimation examples. The methodology followed was to work from concrete examples given in the textbooks, and, from these examples, to identify the concepts of the domain and the relationships between those concepts. Input from domain experts was solicited to validate these efforts. The domain theory is a collection of modular subtheories each containing a set of axioms describing the primitives in the subtheory and the relations between them, expressed in first order logic. *Abstract* primitives encapsulate specification-level concepts in the domain (e.g., fuse data from two sensors) whereas *concrete* primitives define wrappers around implemented software components (e.g., a Kalman filter component).

Figure 4 shows the structure of the subtheories in the current domain theory. The arrows show which subtheories import other subtheories (e.g., the axioms for frame conversions import the NAIF axioms). In the synthesis proofs, the NAIF axioms and frame/coordinate axioms are applied using resolution, paramodulation and demodulation. All other axioms are applied using demodulation only. This was a restriction made to control the proof process. The arrows in Figure 4 also manifest themselves in the axioms: the rules refine primitives from one subtheory into primitives from a subtheory connected by an arrow. Note that the leaf nodes of Figure 4 are subtheories whose primitives appear in the final applicative term. In essence, high-level abstract primitives specifying Kalman filter architectures and sensor configurations are refined into primitives of Euclidean geometry and matrix/vector operations. Refinements also take place *within* the theories that refine primitives of those theories into primitives that are wrappers around code components.

Referring to Figure 1, the specification for a typical example describes the sensor configuration and the Kalman filter architecture. Sensors are specified by giving the datatypes of their outputs — e.g., a DME sensor outputs a distance in a particular frame and coordinate system, and its (noisy) outputs are distributed with a given mean and variance. The Kalman Filter subtheory describes how to derive the necessary matrix inputs (e.g., the Φ and H matrices) to a Kalman filter code component (see Figure 5). Figure 6 gives an axiom from the Kalman filter subtheory for fusing data from multiple sensors using a linearized Kalman filter. Each of the `derive` operations is refined by domain theory axioms into a matrix that is input to the Kalman filter routine.

In general, the synthesis engine applies proof search to apply the axioms in a way that suit the current context. This may involve making pre-defined assumptions as to the nature of the current problem (e.g., that the nominal estimate is close enough to the true value to enable a Taylor series expansion to be accurate) but these assumptions appear explicitly in the final explanations presented to the user.

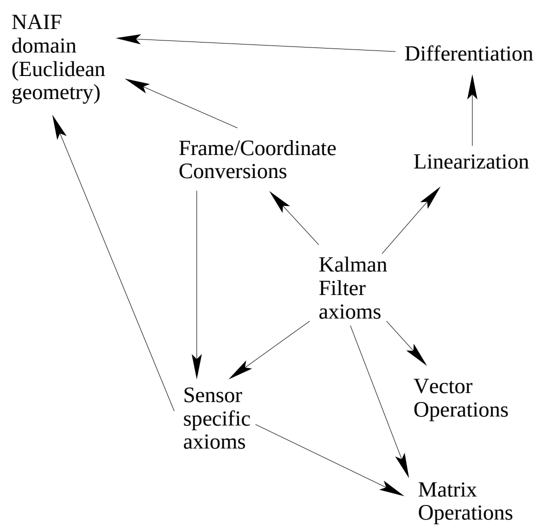


Figure 4. AMPHION/NAV **Domain Theory Organization.**

To give a flavor of the reasoning involved in the synthesis system, consider how the H matrix is formed. Each row in the H matrix corresponds to a measurement and each column to a state variable (so, in the running example, H has 2 rows, one for each DME sensor, and 9 columns, for position, velocity, attitude in 3 directions). Each entry in H is a linearization of the relationship between a measurement and a state variable. In general, the relationship that holds between measurements and variables may not be linear. The domain theory contains a linearization subtheory for applying Taylor series expansion and discarding higher order terms under appropriate assumptions.

In fact, the previous Amphion/NAIF domain theory for geometry can be reused in the linearization process. For example, a DME measurement, z_1, is a distance from the current coordinates to the DME tower, which is expressed very naturally using NAIF's geometric constructs:

$$z_1 = twoPointsToDistance(dmeCoords, coords) \quad (3)$$

Axioms in the domain theory describe how to linearize this equation (it is an abstract representation of the usual Pythagorean distance) around the nominal coordinates. The ease of reuse of the existing NAIF domain theory provides strong evidence that it is possible to develop domain theories in a piece-wise fashion and bring them together in such a way that interactions between theories do not adversely affect the code generation process.

A fully declarative domain theory is ideal for expressing the concepts in a new domain and for communicating and

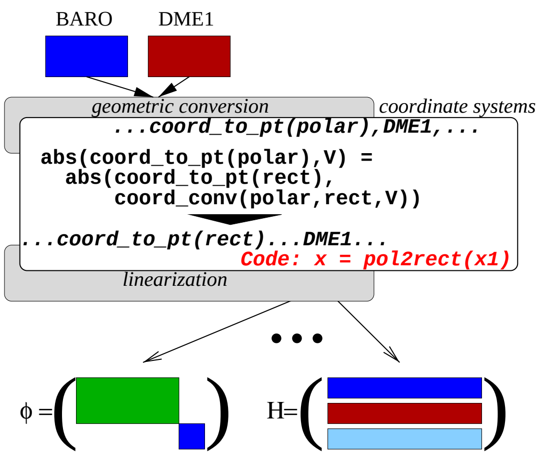

BARO
DME1
geometric conversion
coordinate systems
...coord_to_pt(polar),DME1,...
abs(coord_to_pt(polar),V) =
abs(coord_to_pt(rect),
coord_conv(polar,rect,V))
...coord_to_pt(rect)...DME1...
Code: x = pol2rect(x1)
linearization
φ =
H=

Figure 5. Synthesis in AMPHION/NAV

```
sensor_integration_with_linearize(
                        state nominal aiding) =
  extended_kalman_filter(
      derive-initial-estimate(state)
      derive-state-transition-matrix(state)
      derive-measurement-matrix(state nominal aiding)
      derive-process-covariance-matrix(state)
      derive-measurement-covariance-matrix(aiding)
      derive-observation-vector(aiding)
      derive-initial-covariance-matrix(state)
      )
```

Figure 6. A linearized Kalman filter axiom.

validating their relationships. On the other hand, code generation (regardless of which synthesis engine is used) needs more guidance to be successful. As part of our methodology, we began with a highly declarative domain theory and then extended this theory with operational elements to enable successful refinement. The result is that the subtheories make use of resolution, paramodulation and demodulation, but in a restricted fashion. Inference rules such as resolution and paramodulation provide a great deal of expressiveness but easily lead to exponential blow-up in proof search. Demodulation (rewriting), on the other hand, is tractable but requires the domain engineer to impose a left to right ordering on the axioms. In order to leverage the advantages of both of these techniques, we use resolution/paramodulation in specific, well-defined circumstances, such as converting between different frames and coordinate systems. Most of the domain theory, however, consists of rewrite rules. Paramodulation allows axioms to be applied in a way that introduces a fresh variable into the proof goal. This cannot be done using rewriting because the RHS variables must be a subset of the LHS variables. Paramodulation is very useful when converting two domain objects into a common frame. It allows the decision as to which common frame to use to be delayed until a point in the proof where the instantiation of the fresh variable can be determined from the current proof context. Overuse of paramodulation, however, leads to a search space blow-up and so its use was restricted to the most useful applications.

Another way to limit the search space is to make use of *decision procedures* in the theorem proving process. The idea is to solve appropriate subtasks (over ground terms) that come up in the proof by calls to external routines rather than relying on the proof engine. Amphion/NAIF contained decision procedures for instantiating variables representing frames with the appropriate frame. AMPHION/NAV uses SNARK's procedural attachment mechanism to incorporate decision procedures from the KIF library [5] (list manipulations, numeric manipulations etc.) and procedures for low-level matrix manipulations. By combining theorem proving techniques such as demodulation with specialized (and potentially domain-specific) decision procedures, deductive program synthesis can be scaled up to realistic examples.

5 The ExplainIt! Documentation Generator

In [13], we reported on a technique for explaining deductively synthesized software from traces of the synthesis proofs. Intuitively, an explanation of a statement in a generated program is a collection of explained connections between the variables, functions and subroutines in that statement and objects, relations, and functions in the problem specification or domain theory.

The explanation technique works with the proof derivation of the generated program which is a tree whose nodes are sets of formulas together with substitutions of the existentially quantified variables, and whose arcs are steps in the proof (i.e., they encode the “derived from” relation). Thus, an abstract syntax tree (AST) of the synthesized program and the empty clause is the root of this derivation tree. Its leaves are domain theory axioms and the problem specification. Since the AST and all formulas are represented as tree-structures terms, the derivation tree is essentially a tree of trees.

The explanation generation procedure traces back a position in the abstract syntax tree through the derivation tree extracting *explanation equalities* along the way. These equalities record the links between positions of different terms in the derivation. By reasoning with these equalities, *goal explanation equalities* are derived which relate elements of the generated program with terms in the specification and formulas in the domain theory.

In this paper, we report on our subsequent development of the technique for generating explanations to provide complete traceability. By this, we mean a system that generates an explanation for every executable statement in a synthesized program in vocabulary that a domain expert understands. The explanation indicates why that statement is in the program and how the statement relates to the problem specification and the domain theory. In the following, we describe explanation equalities, explanation templates and their instantiation, and the manner in which the document is assembled.

5.1 Explanation Equalities

From each step in a derivation, a set of explanation equalities is extracted describing the "links" between "pieces" of a formula and its parent formulas with respect to the proof step. Thus, each explanation equality is a logical consequence of the semantics of the inference rule applied in the step and the parent formulas.

The pieces of a formula are identified using a position notation. A position in a formula is specified by a path description from the root of the formula to that position. A path description is a sequence of argument position selectors, e.g., the path [2,1] specifies the position of b in the term $f(a, g(b, c))$. The subterm at a position selected by a valid path description p in the term t is written $t@p$. This positional description is important because lexically identical terms can appear in multiple places in a formula, but are derived through different paths.

When extracting explanation equalities, it is important that we keep track of where the different occurrences of terms originated. To this end, each node in the derivation tree is assigned a unique number. Then, each variable x in node n is annotated by x_n. Non-variable terms (constants, functors) are annotated with node number and position. For example, the formula $P(f(x), g(f(y, x)))$ in node n of the derivation is annotated as $P_{n,[0]}(f_{n,[1,0]}(x_n)), g_{n,[2,0]}(f_{n,[2,1,0]}(y_n, x_n))$

Explanation equalities capturing the links between pieces of a formula are assertions of the form $\Phi_1@p_1 = \Phi_2@p_2$ between annotated terms Φ_1, Φ_2. Explanations are also extracted for substitutions generated in each derivation step in the form of equality assertions $x_n = \Psi_p$, where x_n is the variable x appearing in formula n and Ψ_p is the (annotated) subterm to which x was bound in the inference step, expressed as a position in a formula.

5.2 Templates

Domain theory axioms are annotated with explanation templates. Templates are strings of words and variables. All variables occurring in a template must also occur in the axiom to which the template is attached. Each axiom can have multiple templates each of which is associated with a different position in that axiom. By *templ*(ϕ, p) we denote the template belonging to axiom ϕ at position p. Figure 7 shows the explanation templates for the axiom of Figure 6.

```
templ(extended_kalman_filter,[2]) =
  (Integrate measurements from multiple sensors using an
   Extended Kalman Filter EKF. All measurements are
   assumed to be in the same frame and coordinate
   system. The measurements are linearized around a
   nominal trajectory. Data from aiding sensors is used
   to provide a state estimate based on an underlying
   process/state-vector model. The EKF takes seven
   matrices as input.)
templ(extended_kalman_filter,[2,1]) =
  (an initial state estimate)
templ(extended_kalman_filter,[2,2]) =
  (the state transition matrix Phi)
templ(extended_kalman_filter,[2,3]) =
  (the measurement matrix H)
  ...
templ(extended_kalman_filter,[2,7]) =
  (an initial covariance matrix P_0)
```

Figure 7. Explanation Templates associated with the Kalman Filter Axiom of Figure 6

In that case, we have a template describing the entire axiom. This template is attached at the right-hand side of the equation in Figure 6. Thus, its position is 2. Furthermore, each of the 7 arguments of the extended Kalman filter contain their own explanation templates.

5.3 Template Composition and Instantiation

The composition of an explanation for a position in the generated program (or, for that matter, any position in any formula in a derivation) is constructed from the templates associated with the explanation equalities described above by construction of an equivalence class. Let $\Phi_0@p_0$ be a term in a formula of the derivation. Then, we define $C_E(\Phi_0@p_0) = \{\Phi_i@p_i \mid i = 0, \ldots\}$ as the equivalence class of $\Phi_0@p_0$ containing all terms $\Phi_i@p_i$ induced by the explanation equalities of the derivation.

Then the desired goal explanation equalities linking $\Phi_0@p_0$ to the specification and domain theory are contained in $C_E(\Phi_0@p_0)$. The explanation templates for $\Phi_0@p_0$ can be found in the set $T = \{\textit{templ}(\Phi_i, p_i) \mid \Phi_i@p_i \in C_E(\Phi_0@p_0)\}$ of templates attached to the formula positions in C_E.

To construct the explanation, the templates in this set are instantiated and concatenated together. Each individual template is instantiated by replacing each variable x_Φ with the non-variable term in $C_E(x_\Phi)$ that occurs in the problem specification. Composition of the templates is accomplished by imposing an ordering on the subformulas: for $\Phi_i, \Phi_j \in C_E(\Phi_0@p_0)$ such that Φ_i occurs earlier in the derivation than Φ_j then *templ*(Φ_i, p_i) precedes *templ*(Φ_j, p_j) in the explanation.

As an example, the DME example from Section 2 contains the following subterm in the applicative term gener-

```
<function name=f>
<reason axiom=...> Explanation returned for entire term
</reason>
   <argument type =...>
      XML explanation returned for the term $t_1$
    </argument>
   ...
   <argument type =...>
     XML explanation returned for the term $t_n$
    </argument>
</function>
```

Figure 8. XML representation of the explanations for a term $f(t_1, ..., t_n)$.

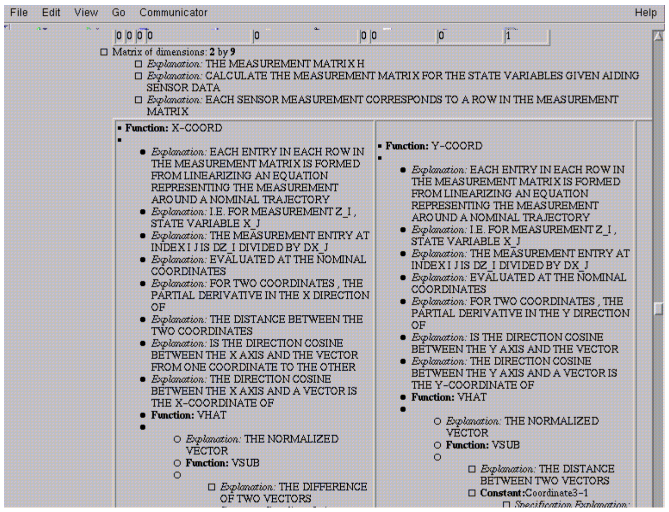
ated:
`mk-mx`(2,9, `list(list(`
`x-coord`(`vhat`(`vsub`(
`Coordinate3-1`,`Coordinate-Dme1-Posn`)
...)))

This subterm computes the measurement matrix, H. `x-coord(...)` is one entry of the 2×9 matrix that computes the linearized relationship between the state variable representing position in the x direction, and the measurement variable representing the measurement from the first DME sensor. Figure 9 gives the explanation for this subterm. Each bullet comes from an explanation template. The first four bullets describe in a generic way the contents of the element at position $(1, 1)$ in the matrix. The current output text is somewhat rough but tells the user that the entry represents $\frac{\partial z_1}{\partial x_1}$ evaluated using the nominal values for $\mathbf{x}$. The next three bullets describe the application of axioms that rewrite this partial derivative in terms of a direction cosine. The remaining bullets explain that the direction cosine is equivalent to the function composition `vhat(vsub(...))`.

5.4 Document Assembly

The final output of ExplainIt! is a document which explains each part of the synthesized code in a format suitable for the domain engineer. The structure of the explanation is reflected in the computational structure of the applicative term. Thus, explanations are constructed for each position in the applicative term. As a flexible intermediate format, XML is being used, because it facilities the generation of various document formats and the use of hyper-links allow to the user transparently trace between the final code and the explanation document. This is necessary, because the structure of the code does not necessarily coincide with the structure of the applicative program. For each subterm $f(t_1, ..., t_n)$ a structured XML data structure is generated as shown in Figure 8.

XSLT [8] is used to produce the final version of the explanation from this XML document. XSLT transforms the



☐ Matrix of dimensions: **2** by **9**
- ☐ *Explanation:* THE MEASUREMENT MATRIX H
- ☐ *Explanation:* CALCULATE THE MEASUREMENT MATRIX FOR THE STATE VARIABLES GIVEN AIDING SENSOR DATA
- ☐ *Explanation:* EACH SENSOR MEASUREMENT CORRESPONDS TO A ROW IN THE MEASUREMENT MATRIX

▪ **Function:** X-COORD
▪
- *Explanation:* EACH ENTRY IN EACH ROW IN THE MEASUREMENT MATRIX IS FORMED FROM LINEARIZING AN EQUATION REPRESENTING THE MEASUREMENT AROUND A NOMINAL TRAJECTORY
- *Explanation:* I.E. FOR MEASUREMENT Z_I , STATE VARIABLE X_J
- *Explanation:* THE MEASUREMENT ENTRY AT INDEX I J IS DZ_I DIVIDED BY DX_J
- *Explanation:* EVALUATED AT THE NOMINAL COORDINATES
- *Explanation:* FOR TWO COORDINATES , THE PARTIAL DERIVATIVE IN THE X DIRECTION OF
- *Explanation:* THE DISTANCE BETWEEN THE TWO COORDINATES
- *Explanation:* IS THE DIRECTION COSINE BETWEEN THE X AXIS AND THE VECTOR FROM ONE COORDINATE TO THE OTHER
- *Explanation:* THE DIRECTION COSINE BETWEEN THE X AXIS AND A VECTOR IS THE X-COORDINATE OF
- **Function:** VHAT
-
    - ○ *Explanation:* THE NORMALIZED VECTOR
    - ○ **Function:** VSUB
    - ○
        - ☐ *Explanation:* THE DIFFERENCE OF TWO VECTORS

▪ **Function:** Y-COORD
▪
- *Explanation:* EACH ENTRY IN EACH ROW IN THE MEASUREMENT MATRIX IS FORMED FROM LINEARIZING AN EQUATION REPRESENTING THE MEASUREMENT AROUND A NOMINAL TRAJECTORY
- *Explanation:* I.E. FOR MEASUREMENT Z_I , STATE VARIABLE X_J
- *Explanation:* THE MEASUREMENT ENTRY AT INDEX I J IS DZ_I DIVIDED BY DX_J
- *Explanation:* EVALUATED AT THE NOMINAL COORDINATES
- *Explanation:* FOR TWO COORDINATES , THE PARTIAL DERIVATIVE IN THE Y DIRECTION OF
- *Explanation:* IS THE DIRECTION COSINE BETWEEN THE Y AXIS AND THE VECTOR
- *Explanation:* THE DIRECTION COSINE BETWEEN THE Y AXIS AND A VECTOR IS THE Y-COORDINATE OF
- **Function:** VHAT
-
    - ○ *Explanation:* THE NORMALIZED VECTOR
    - ○ **Function:** VSUB
    - ○
        - ☐ *Explanation:* THE DISTANCE BETWEEN TWO VECTORS
        - ☐ **Constant:**Coordinate3-1
            - ☐ *Specification Explanation:*

Figure 9. Screen dump of a part of the explanation document

function-argument tag structure in the XML document into HTML with a pargraph containing the explanation of the function followed by a bulleted list of the explanations of the arguments. It further processes terms whose head is `mk-mx` into tables. The resulting document for the term `mk-mx(2,9,...)` in the applicative term from the previous section is shown in Figure 9.

This XSLT parser can easily be adapted to handle various syntactic transformations. Thus, the basic structure of ExplainIt! can be adapted to other domains. ExplainIt! is thus configurable in a similar way like Hallgren's Proof Editor [7], or the ILF system [3].

6 Related Work

Over recent years, automatic code generation has become a hot topic in various industries, including the aerospace industry. A number of commercial tools are now available that carry out domain-specific code generation. MatrixX[3] and Simulink[4] provide graphical languages for modeling dynamic systems from which code can be generated. In essence, however, these code generators are little more than domain-specific compilers. The domain-knowledge embedded in these systems comes from the fact that each graphical construct corresponds to a domain-specific primitive and can be compiled in a 1-to-1 fashion into code. The mapping in AMPHION/NAV is not 1-to-1. The code generation process may involve reasoning over

[3]from Wind River Systems
[4]from MathWorks

deep domain knowledge and may make assumptions about the current problem context that will affect the code produced. It is also notoriously difficult to integrate code generated using MatrixX or Simulink with hand-written code as there is no traceability between model and code. AMPHION/NAV overcomes this problem through its explanation mechanism. As further work, we intend to investigate the problem of optimizing the generated code. The domain theory in AMPHION/NAV should enable domain-specific optimizations that could not be achieved using MatrixX or Simulink.

7 Conclusions

We have presented AMPHION/NAV, a deductive synthesis system for the automatic generation of state estimation software with Kalman filters. We have used this system to synthesize roughly a dozen graduate-level textbook examples of single-mode geometric state estimation software and variants thereof. The examples use either an inertial navigation system (INS, see our example in the text), or a GPS system as its basis. We have used models for distance measuring equipment (DME), VOR (measuring the angle between the aircraft and a fixed station), GPS, and baro-altimeter. In these examples, SNARK could find a proof within a few minutes.

Although there have been many improvements over the old synthesis system with respect to domain complexity, usability, and generation of explanations, there is still a number of important issues to be addressed. During development of AMPHION/NAV it turned out that the graphical specification language (which is translated into first-order formulas) is not well suited for the state estimation domain. Many examples require a very large graphical specification, resulting in a low leverage factor between size of specification and size of the synthesized code. Furthermore, a concise semantics for the specification language is required. For practical usability, we will work together with domain experts on the development of a practical, yet concise specification language.

As described in the paper, the development of the domain theory turned out to be a central issue for our synthesis system. Although the old NAIF domain theory could be reused in an as-is manner, a fact which nicely demonstrates compositionality of domain theories, structure and development process for the domain theory needs to be improved substantially. In particular, for safety-critical domains, a careful development process for the domain theory is essential, because proving correctness for an entire domain theory is practically impossible. Current work aims to develop a domain theory in a much more structured and fundamental way. We are investigating in how far techniques from object-oriented software design can be of help for our task. Such a principled development, which is embedded in a highly iterative design process also addresses issues of interaction between the domain theory and the deductive machinery. To avoid unnecessary search spaces, which is an important prerequisite for scalability, parts of the domain theory which cope with calculations or assembly of data structures (e.g., matrices) should employ decision procedures or procedural attachment to the theorem prover. A highly structured domain theory allows us to clearly identify such parts.

Within the paradigm of deductive program synthesis, all information required to construct the program is contained in the proof. Our experience with practical program synthesis, however, showed that synthesizing code alone, i.e., without detailed documentation is not enough. Hence, in developing AMPHION/NAV, much effort was spent on the explanation system. Automatic generation of documentation is only a first step. Future work will investigate how far deductive synthesis can support computer-supported certification of safety-critical code by automatic generation of verification proof obligations, invariants, and other annotations for the synthesized code which then can be checked by a small and trusted proof checker.

References

[1] R. G. Brown and P. Hwang. *Introduction to Random Signals and Applied Kalman Filtering.* John Wiley & Sons, 3rd edition, 1997.

[2] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications.* Addison Wesley, 2000.

[3] B. I. Dahn and A. Wolf. *Natural Language Presentation and Combination of Automatically Generated Proofs,* volume 3 of *Applied Logic Series,* pages 175–192. Kluwer Academic Publishers, 1996.

[4] A. Gelb, editor. *Applied Optimal Estimation.* MIT Press, 1974.

[5] M. Genesereth. Knowledge interchange format, 1998. URL: `http://logic.stanford.edu/kif/kif.html`.

[6] C. C. Green. Application of theorem proving to problem solving. In *Proceedings Intl. Joint Conf. on Artificial Intelligence,* pages 219–240, 1969.

[7] T. Hallgren and A. Ranta. An extensible proof text editor. In *Logic for Programming and Automated Reasoning (LPAR'2000),* volume 1955 of *LNAI,* pages 70–84. Springer-Verlag, 2000.

[8] M. Kay. *XSLT Programmer's Reference.* Wrox Press, 2000.

[9] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering,* 18(8):674–704, 1994.

[10] G. M. Siouris. *Aerospace Avionics Systems: A modern Synthesis.* Academic Press, Inc., 1993.

[11] M. Stickel. The snark theorem prover, 2001. URL: `http://www.ai.sri.com/~stickel/snark.html`.

[12] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In A. Bundy, editor, *Proc. 12th Conference on Automated Deduction*, Nancy, France, June 1994. Springer Verlag LNCS 814.

[13] J. van Baalen, P. Robinson, M. Lowry, and T. Pressburger. Explaining synthesized software. In *Thirteenth International Conference on Automated Software Engineering*, pages 240–248. IEEE Computer Society Press, 1998.